APPENDIX B. Moso Process Manager

Moso is a prototype process manager for the SSS environment. It was created to solve the disjoint relationship between the Fountain node monitor and MPD process manager, which is explained in detail in section 4.1. Its role as a process manager is critical since it provides the services to start, stop, and signal user processes as requested by other components such as the scheduler or queue manager. To accomplish this, Moso uses the concept of a process group, where a single process group consists of `n` processes running on `n` hosts in the cluster. This concept is somewhat similar to the POSIX definition of a process group, with the exception that a Moso process group is split across nodes in a cluster.

To manage user processes, Moso uses one Moso daemon process per node in the process group. The role of a Moso daemon is similar to that of the managers in the MPD process manager [7]. It collects standard output and error from the user process, forwards signals, and monitors the exit status. To collect standard output and error in a scalable fashion, the Moso daemons are arranged in a binary tree topology, shown in Figure B.1. The large black circles represent Fountain daemons, configured in a binary tree topology represented by the larger solid black lines. They can fork and manage multiple Moso daemons concurrently. The dashed vertical lines represent pipes used for communicate between a Fountain daemon and its forked Moso daemon. The smaller grey circles with numbers representing their process group are the Moso daemons. They each have a single pipe to their user process, represented by another vertical dashed line. The smaller solid grey lines between the Moso daemons represent their binary tree topology within their process group. In Figure B.1 there is a total of 7 Fountain daemons, 12 Moso daemons, and 3 process groups. Note that the master Fountain daemon is not capable of forking a Moso daemon in our design.

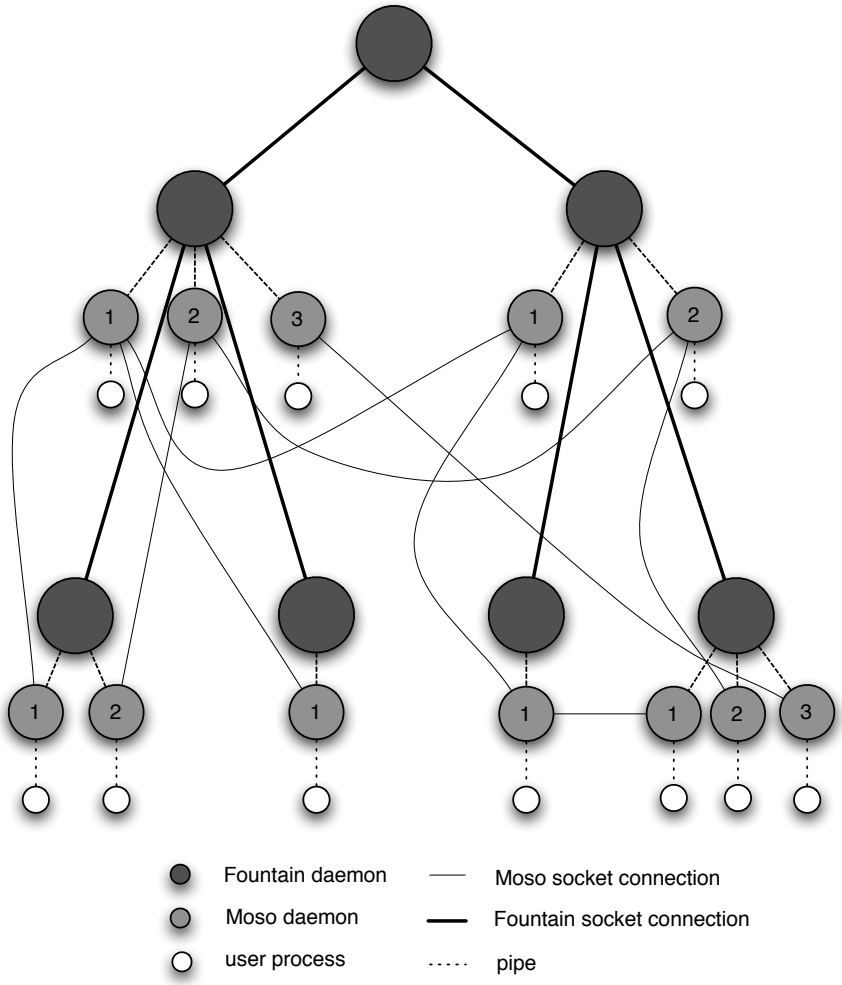

Figure B.1 Fountain daemons, Moso daemons, and user processes

A Moso daemon is created when a process group is created, exists for the lifetime of the process group, and terminates when the process group finishes. Since they only exist for the lifetime of their process group, Moso requires the help of Fountain to create and destroy Moso daemons in order to achieve decent scalability. Moso itself maintains an internal container of process group identifiers and their associated data, such as state, list of hosts, and any other pertinent information. The following sections will explain how Fountain handles requests from Moso to create, signal, and kill process groups.

Creating a Process Group

When the cluster scheduler decides its time to run a user's job, it will inform the queue manager, which will request a process group be created for that job. To maintain compatibility with other components in the SSS environment, Moso uses the Less Restrictive Syntax (LRS) for XML messages. The following is an example request to create a process group on 5 hosts:

```
<create-proces-group>
  <submitter>samm</submitter>
  <process-spec>
    <env name="PWD">/Users/samm/moso</env>
    <exec>hostname</exec>
  </process-spec>
  <host-spec>m0
  m1
  m2
  m4
  m5
  </host-spec>
</create-process-group>
```

When Moso receives such a request, it creates a process group object for internal bookkeeping purposes and forwards the request to the master Fountain daemon. Upon receiving the request, the master Fountain daemon performs the following tasks:

1. Check the contents of the `host-spec` element to ensure each hostname belongs to a valid host in the Fountain tree topology.

2. Translate the request into an SSSRMAP formatted message and send it to its child Fountain daemons.

3. Wait for responses from its child Fountain daemons indicating if they successfully forked a Moso daemon.

4. Collect Moso daemon listen ports from the response messages, assemble them into a list.

5. Send a message containing the list of Moso daemon listen ports to its child Fountain daemons.

6. Wait for responses from its child Fountain daemons. Inform Moso if the process group was created or not.

The first task in the above list already solves the disjointness problem we experienced with Fountain and the MPD process manager. If Moso attempts to create a process group with a host that does not exist in the Fountain tree topology, the master Fountain daemon will respond with an error. If all the hosts in the requested process group are found in the topology, we can assume each node is ready to run a user job. Additional work remains to coordinate the list of hosts in the tree topology with each node's state that is maintained by the Fountain server. A system administrator can set a node's state to `Unavailable` through the Fountain administrative interface. While it is possible a node with a state of `Unavailable` can have a Fountain daemon running on it, any requests to create a process group containing any host with a state of `Unavailable` should be rejected by the master Fountain daemon. In practice, this event should rarely happen since the queue manager will only request a process group be created after it has been told to do so by the cluster scheduler. The cluster scheduler will only schedule the user's job when all the requested hosts have an acceptable state.

When the slave Fountain daemons receive a create process group request, they first forward the request to their child Fountain daemons. Next, they match their hostname against the `host-spec` element. If a match is found, they fork a Moso daemon. If an error occurred during the fork system call (ex: process table full) an error response is sent to the Fountain daemon's parent. Before forking a Moso daemon, a Fountain daemon opens a listening port for it to inherit. The `fork` system call provides the semantics for child processes to inherit all open file descriptors of their parent. We chose this design because during testing it proved to be too time consuming to allow a Moso daemon to open its own listening port, then send this port number to its Fountain daemon. On several compute nodes under high load, we measured times in excess of 10 seconds to just fork a Moso daemon. This amount of time is insufficient for the master Fountain daemon's purposes, which uses timeout values while waiting for response messages from its children after sending them a create process group request.

After forking a Moso daemon, the Fountain daemon collects a listening port message from each of its children and appends their information into a message containing its Moso daemon's listening port. Eventually, the master Fountain daemon will receive all of these messages and collect them into a single message, then send it down the tree topology. This process is required because the forked Moso daemons need to know which parent daemon to connect to, and it is not always the same parent as its Fountain daemon (see Figure B.1). When the slave Fountain daemons receive the message containing all of the Moso daemon listening ports, they find their Moso daemon in the list and pick its parent node for the binary tree topology using equation B.1. They then send a success or failure response up the Fountain tree topology where the master daemon collects them.

$$parent = \lfloor (pos - 1)/2 \rfloor \tag{B.1}$$

After the master Fountain daemon receives all the responses from the slave daemons, it checks if the requested hosts in the `host-spec` element forked Moso daemons. If every host requested actually forked a Moso daemon, a success response is sent to Moso containing the list of hosts in the process group. Moso then adds this information into a process group container.

The act of starting the user job is controlled by the Moso daemons themselves. The root Moso daemon is aware of how many daemons will be present in its binary tree topology when it's fully formed, so it will periodically send a `trace` request message to its children. Its children will forward that request to their children, then collect their responses and add their own hostname before sending a trace response to the root Moso daemon. Only after the correct number of Moso daemons are found in the topology will the root Moso daemon send a `start` request message to its children. This procedure is shown in Figure B.2. It is somewhat similar to a `MPI_Barrier` collective used by most Message Passing Interface implementations.

The reason we chose to push the role of starting the user job into the domain of the Moso daemons is similar to our decision to have a Fountain daemon open its Moso daemon's listening port before forking it. It proved to be too time consuming during testing to wait until each forked Moso daemon had connected to its selected parent daemon. The time required to handle

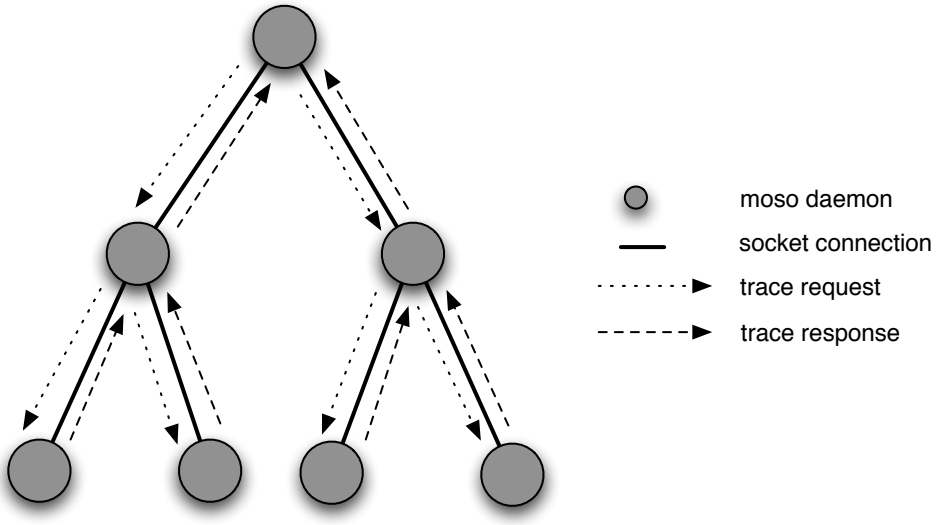

Figure B.2 Trace request and response messages sent by Moso daemons

a create process group request is much faster using these two methods.

Signaling a Process Group

When the queue manager has determined a user job has run over its allocated time limit, it will ask the process manager to deliver various signals to the process group. Some process manager implementations only deliver signals to the first process in the process group. They rely on the user job being responsible for propagating the signal to the other user processes. Moso's implementation sends the signal to each user process in the process group. The request received by Moso looks similar to this:

```
<signal-proces-group>
  <signal>SIGTERM</signal>
  <process-group>
    <pgid>2</pgid>
  </process-group>
</signal-process-group>
```

Moso performs some semantics checking on the request, such as enforcing a valid process group id, and a valid signal type, then sends the request to the master Fountain daemon. It does not

expect a response, similar to how the UNIX `signal` command does not return a value, unless the process does not exist in which case it returns -1.

When the master Fountain daemon receives the signal process group request it translates the request into an SSSRMAP message and forwards it to its child daemons. The child daemons first forward the request to their children, then match the process group id to any Moso daemons they have forked. If a match is found, the signal is delivered. If no match is found, the request is ignored.

Killing a Process Group

A process group is killed after all of its client processes have exited. At that point, each Moso daemon has served its purpose and can be safely killed without affecting any other process. The problem here is detecting when all the user processes have exited. To accomplish this, the Moso server will periodically send a request message to query each Fountain daemon about its Moso daemons exit status. Only after all of the user processes in the process group have returned an exit status, will the Moso server send a kill process group request formatted like so:

```
<kill-proces-group>
  <process-group>
    <pgid>2</pgid>
  </process-group>
</signal-process-group>
```

Upon receiving this request, the master Fountain daemon translates it into an SSSRMAP message and forwards it to its children. The slave Fountain daemons first forward the request to their children, then match the process group id element to their list of Moso daemons. If a match is found, the Moso daemon is killed. Alternatively, Moso can kill a process group after being requested to do so by a user. The syntax for this request is similar to what is shown above.

Results and Discussion

We tested the Moso process manager on Scink, a cluster which is described in detail in Chapter 6. The Fountain daemons were arranged in a binary tree topology, with one, two, or four daemons running on each of the 64 compute nodes in the cluster. Measuring parallel job performance is a difficult task since it can depend on a great deal of parameters. For our purposes of assessing performance of a process manager, we desired to know the time it takes to create the process group, as well as the time it takes the Moso daemons to connect themselves into a binary tree topology.

The time taken to create a process group begins when the master Fountain daemon receives the request from Moso, processes it, and ends when it sends a success or failure response back to Moso. We expect this time to be somewhat similar to the time required for the Fountain server to query a similarly configured topology, except in this case we are forking another process (a Moso daemon) instead of collecting node monitoring information. Based on our results in Chapter 6, we expect this number to stay somewhat constant regardless of the number of hosts in the process group. This is because the create process group message has to be propagated to each Fountain daemon in the tree topology, even if the request only asks to create a process group consisting of a single host.

The time taken for the Moso daemons to bootstrap themselves into a binary tree topology begins when the root Moso daemon enters its main loop, and ends after it has determined the correct number of Moso daemons have joined. This time will probably vary wildly depending on the status of the hosts in the process group. If certain nodes in the cluster are running at near peak utilization, it may take their Moso daemon significantly longer to join the tree topology than nodes that are idle. Table B.1 and Figure B.3 show the results from testing Moso when creating four differently sized process groups on Scink. The numbers shown represent an average of three separate test runs.

The results are similar to what we expected. The time required to create a process group scales linearly with the number of Fountain daemons, requiring under 0.5 seconds to setup a process group of 64 hosts on all three configurations of Fountain daemons. The time to setup

Table B.1 Elapsed process group creation time on Scink

Fountain daemons	Hosts	Create Process Group (ms)	Bootstrap Binary Tree (sec)
65	3	280.71	6.26
65	9	285.84	7.46
65	27	301.03	13.46
65	64	315.93	23.80
129	3	354.54	5.64
129	9	353.75	7.40
129	27	360.71	19.23
129	63	426.71	23.49
257	3	403.52	5.79
257	9	437.21	30.67
257	27	404.12	30.59
257	63	458.27	22.60

a binary tree for the Moso daemons to communicate however, requires more time as the size of the process group is increased. This time seems to be somewhat random, and is probably related to the overhead of the Scink nodes in the process group. For example, when testing a 64 host process group created with 257 Fountain daemons, it required slightly over 20 seconds for the Moso daemons to create their tree topology. However, with the same configuration of Fountain daemons, it required over 30 seconds to create a process group of both 9 and 27 hosts. We expect this time to decrease when the compute nodes in the process group have utilization values closer to 0.

Summary

This appendix has shown how Fountain can provide the required process management services to a prototype SSS process manager called Moso. It accomplishes these tasks by forking Moso daemons to manage user processes. The Moso daemons in turn, use information from their parent Fountain daemon to bootstrap themselves into a binary tree topology, which

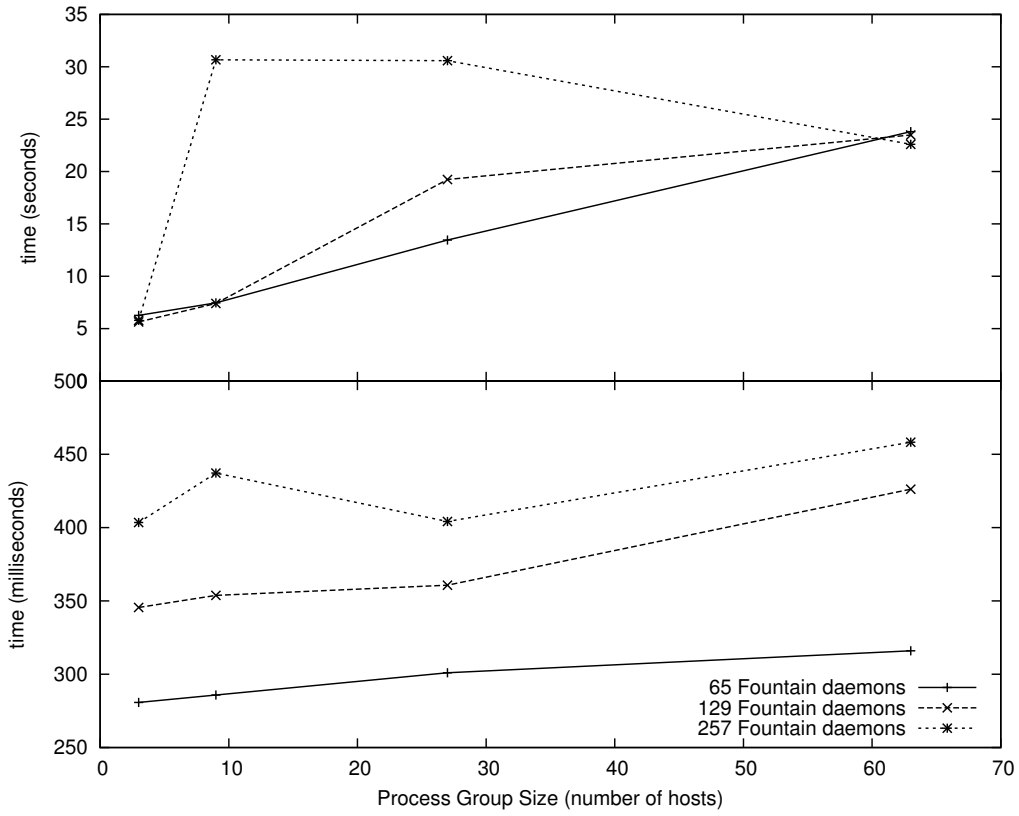

Figure B.3 Process group creation (lower) and binary tree (upper) on Scink

is used to forward standard output, error, and signals to and from the user processes in a scalable fashion. Once all the user processes in the process group have finished, the Moso daemons are killed to prevent unnecessary resource leaks. By combining Fountain and Moso together we have solved the disjoint node monitor and process manager problem experienced with the MPD process manager.

This project is by no means complete. Future work exists to properly incorporate advanced process management features, such as interactive jobs, attached debuggers, different executables for certain ranks, and incorporating parallel library functionality such as MPI.